Zircuit

Security Assessment

February 15th, 2024 — Prepared by OtterSec

Woosun Song procfs@osec.io

Matteo Oliva matt@osec.io

Table of Contents

01 — Executive Summary

Overview

Zircuit Labs engaged OtterSec to assess the `zkr-staking` program. This assessment was conducted between February 13th and February 15th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

In particular, we identified an issue due to using an incorrect Wrapped Ether interface (OS-ZKR-ADV-00), resulting in Ether deposits failing unconditionally. Furthermore, we addressed an issue where the fund provider can be set to a user-specified address, potentially seizing approved funds (OS-ZKR-ADV-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/zircuit-labs/zkr-staking. This audit was performed against commit 6b9eeb4.

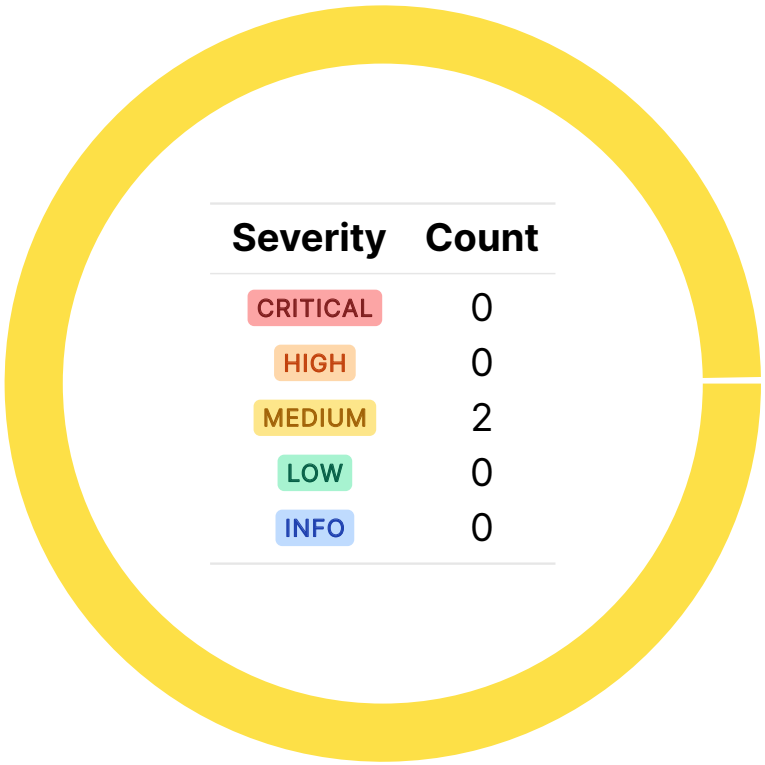
A brief description of the programs is as follows:

Name	Description
zkr-staking	A multi-token staking pool where off-chain processes handle point calculations. Events are emitted upon deposit, withdrawal, and migration, facilitating accurate point computation with the objective of encouraging users to authorize the migration of their liquidity to Zircuit Layer 2.

03 — Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	0
INFO	0

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-ZKR-ADV-00	MEDIUM	RESOLVED	`depositETHFor` utilizes an incorrect signature for `IWETH.deposit`, resulting in the function to fail unconditionally.
OS-ZKR-ADV-01	MEDIUM	RESOLVED	`depositFor` calls `transferFrom` with a user-specified address, when it should be `msg.sender` instead.

Incorrect Function Signature MEDIUM OS-ZKR-ADV-00

Description

`depositETHFor` fails unconditionally due to an incorrect utilization of `IWETH.deposit`. The signature of `IWETH.deposit` should be `deposit() external payable`. However, the signature `deposit(uint256) external payable` is used instead.

>_ contracts/ZtakingPool.sol solidity

```
function depositETHFor(address _for) whenNotPaused payable external {
    if (msg.value == 0) revert DepositAmountCannotBeZero();
    if (_for== address(0)) revert CannotDepositForZeroAddress();
    if (!tokenAllowlist[WETH_ADDRESS]) revert TokenNotAllowedForStaking();

    balance[WETH_ADDRESS][_for] += msg.value;
    emit Deposit(++eventId, _for, WETH_ADDRESS, msg.value);

    IWETH(WETH_ADDRESS).deposit{value:msg.value}(msg.value);
}
```

This oversight results in `depositETHFor` always reverting, effectively blocking all Ether deposits. Consequently, address this issue in order to keep the staking pool live.

Remediation

Remove the `uint256` argument from the function signature of `IWETH.deposit`.

Patch

Fixed in 05cf7f8.

Incorrect Fund Provider MEDIUM OS-ZKR-ADV-01

Description

The existing `depositFor` implementation is not accurate, as it withdraws funds from an address specified by the user rather than from `msg.sender`. This poses an issue as its behavior deviates from that of `depositETHFor`, which receives funds through `msg.value`, inherently taking funds from `msg.sender`. Moreover, utilizing `transferFrom` with a user-defined `from` parameter is risky as it allows potential attackers to gain control over approved funds.

>_ *contracts/ZtakingPool.sol* solidity

```
function depositFor(address _token, address _for, uint256 _amount) whenNotPaused external {
    if (_amount == 0) revert DepositAmountCannotBeZero();
    if (_for== address(0)) revert CannotDepositForZeroAddress();
    if (!tokenAllowlist[_token]) revert TokenNotAllowedForStaking();

    balance[_token][_for] += _amount;

    emit Deposit(++eventId, _for, _token, _amount);

    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
}
```

This error allows an attacker to forcefully transfer funds from any user who has given approval to the pool. However, the security impact is limited to user inconvenience as users may withdraw funds post-incident.

Remediation

Set the `from` argument to `msg.sender` instead of `for`.

Patch

Fixed in 05cf7f8.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.